The Atari ST M68000 tutorial part 3 – of various things mystic and important, mainly concerning the art of understanding digits and performing traps

Hello again! I've gotten some positive feedback on the first two tutorials, so I'm glad to begin writing this third one. As promised, I'll try to explain how computers think when it comes to numbers, the layout of the Atari hardware which will guide us to the workings of traps. I bet a very few understood anything about that.

So, now I'll try to explain what may have been a bit lofty in the previous two parts of the tutorial; how we express numbers. When you see three rocks, you count them, one, two, and three. We have speech in order to communicate our thoughts and emotions to other people, and we have writing in order to communicate speech in written form. We use numbers to communicate "counting". We have chosen the symbol '3' to express the amount you reach when counting one, two and three. However, this value, this "there are three things of something", can of course be written in different ways.

Our number system is based on base 10, meaning that we have ten different symbols to express values, one of them being no values (also known as zero), which leave us with the ability to count to nine. Once the number nine has been reached, we need to start using numbers over again, we don't have a symbol for the value ten, so we have to combine the numbers we have in some way in order to express this. What we do is to say that different positions are "worth" different. For example, in the expression 23, the number 2 is worth ten times as much as three. Do we see a connection here? We use base 10, each number is worth ten times as much as its predecessor. In the expression 123, 1 is worth ten times as much as 2, and one hundred times as much as 3. To calculate the value of the expression 123, we really use this formula

(1 * 10 ^ 2) + (2 * 10^1) + (3 * 10^0)

the generic formula looks like this

value * base^position …

OK, you say hesitantly, perhaps I understand something of what you're trying to say; now what about computers? Computers use base 2, they are binary and can only count 1's and 0's. There is either current through a circuit, or there isn't. They only use two symbols to express values. As you can imagine, the result is very long expressions, like 10101010101111010001010 10. For presenting the value three, the computer uses the symbols 11. Using our generic formula above, we translate this to

1 * 2^1 + 1 * 2^0 = 3

For expressing the number five, we would get 101

1		0		1	
1 * 2^2	+	0 * 2^1	+	1*2^0	= 5

4 + 0 + 1 = 5

Think of it as tumblers, when we have base 10, the tumbler starts at zero, make nine “ticks”, then the tumbler directly to the left makes one tick displaying one, the first tumbler reaches zero again, make nine ticks, the left tumbler makes one tick, reaching two, the first tumbler starts at zero again. When the second tumbler is at nine, and the first tumbler is also at nine, the next tick will take both tumblers back to zero, and a third tumbler, the one directly to the left of the second tumbler, will go from zero to one, and so forth. If we have base 2, there are only two numbers on the tumblers, 1 and 0.

In order to facilitate things, we also have base 16, or hexadecimal numbers. The reason for this is that conversion between decimal (base 10) and binary (base 2) numbers are somewhat cumbersome, while conversation between hexadecimal, or hex for short (base 16) and binary is fairly easy. We use binary because the computer thinks in binary, we use hex because it’s easier to convert binary into hex rather than decimal, and hex is much more manageable (less symbols required to express the same value), and of course we use decimal because we think in decimal.

If you’ve understood what I said above, you’ll wonder how on Earth we use hexadecimal values, since we would need six more symbols for expressing the values ten, eleven, twelve, thirteen, fourteen and fifteen. We simply use letters from our alphabet. A = ten, B = eleven, C = twelve, D = thirteen, E = fourteen and F = fifteen. So to express the value fifteen in hex, we simply use ‘F’. To express sixteen, we get 10 (1*16^1 + 0*16^0). Seventeen is 11 (1*16^1 + 1*16^0. AF would mean one hundred and seventy five (10*16^1 + 15*16^0) and finally F5A would be three thousand nine hundred and thirty (15*16^2 + 5*16^1 + 10*16^0). If you don’t get all this, this topic will probably be covered by every beginners book and tutorial out there, go find an additional source of information.

In Devpac, values are expressed by putting ‘#’ in front of a number, then further using ‘$’ or ‘%’ before numbers to indicate hexadecimal or binary numbers. Thus, ‘#’ means decimal, ‘#$’ means hexadecimal value and ‘#%’ means binary value. So, as you see, it’s customary to express memory addresses in hexadecimal. ‘$’ (without ‘#’) means value at memory address.

Huh, that was that, now onto the architecture of the Atari! Every computer comes with a processor of different (or similar) brands, the Atari has a M68000 processor, so does the Amiga 500. The PC has an 8086 and so on. The processor comes with a different set of instructions and ways of working. This is the real challenge of assembly programming, you must know how the processor behaves that you are trying to program. In high level languages, such as Pascal and C, the programming language usually handles the processor dependent parts, and the code is much more portable between different platforms (computers). Assembly programming however, deals with processor level instruction, which makes it much more powerful and controllable. What the high level languages do, is to take the source code and “translate” it into low level (assembly) language, which of course brings waste, since we can tailor make our programs in assembly, while high level languages have to be more generic (since they weren’t tailor made for a specific processor).

This is a good time to acquire a set of M68000 instructions, you will need to refer to these, also a good guide to the ST's hardware would be most useful, the ST Internals for example. With this tutorial should be a listing of the M68K instruction set, called INSTRSET.TXT. In order to explain the GEMDOS trap routine we used in the last tutorial, I'll have to explain registers. The M68K (K here stands for kilo, meaning 1000, thus M68K and M68000 are synonyms) have 16 registers plus one extra; eight data registers and eight address registers plus a program counter. The program counter is really an address register that points on the next instruction to be performed, simply put, it keeps track of the program execution so that the Atari knows where it's at.

The eight data registers work as eight variables that can store data. The eight address registers are used to store addresses, addresses work like pointers to larger chunks of data, as explained in tutorial one. Here is a usage example of the data register.

```
move.w    #10,d0     put value 10 into data register zero
move.w    #5,d1      put value 5 into data register 1
add       d1,d0      adds d1 to d0 and stores value in d0
                     d0 now holds value 15
```

The address register seven (a7) is worth special mentioning, this is the so called stack. The stack is a lovely pile of data that are used in many ways, for example by the trap instructions. It's of special importance to the PC programmer, since the 8086 have very few registers, the stack is used as a temporary variable, the M68K however is equipped with many registers making the usage of the stack as a temporary variable a bit unnecessary (I hope that no super smart programmer reads this and thinks; what an idiot).

OK, on to the traps. There are three parts of the Atari that can perform traps; the GEMDOS, the BIOS and the XBIOS. The GEMDOS is the hardware independent part of the operating system, this means that the GEMDOS will work on different set of hardware on the Atari. The BIOS is concerned with input and output, like keyboard input and so on, it works between the GEMDOS and the hardware. The XBIOS handles the extended features of the Atari hardware, whatever that means.

You call traps by putting the correct information on the stack, and then calling the correct trap number and then cleaning up the stack. First, we need a special number, called a function number, indicating what function we want, for example, going into super mode is number 32 of the GEMDOS, meaning that we have to put the number 32 on the stack, and then call trap #1 (which is the trap number associated with GEMDOS). Usually, you also need to pass additional information to the Atari, this information is put on the stack before the trap number, and then you call the trap. Thus, put any information associated with the function on the stack, then put the function code on the stack and then call the trap number that corresponds to the handler of the function.

It feels a bit confusing with traps and trap numbers, I'll try to sort it out. The BIOS, XBIOS and GEMDOS each have several special instructions, that aren't in the processor, and do special things. Since they all three have several functions, all three will have the function numbered 1. In the BIOS,

function 1 is a function for returning the input device status, in the GEMDOS, function 1 gets a single character from the keyboard, and in XBIOS, function 1 will save memory space. This information must be gathered by referring to a list of all traps available. In the ST Internals, there is a list of all the traps available, and instructions on how to use them.

What you do is to specify what trap function you want by putting information on the stack, then you call either BIOS, XBIOS or GEMDOS, and let them do something with the information. As an example, I give you this alternative way of changing the background colour. The "-(a7)" means put on stack, we'll cover that in the next tutorial.

```
move.w    #$700,-(a7)          colour red
move.w    #0,-(a7)     in colour 0, background colour
move.w    #7,-(a7)     function 7; Setcolor
trap      #14          call XBIOS
addq.l    #6,a7        clean up the stack
```

The first two move instructions put information regarding the call on the stack, first the colour is passed, next the palette number to be changed. Then, the trap instruction number (opcode) must be put on the stack, next we call the XBIOS to process the information. Lastly, the stack needs to be cleaned up so that none of the information we entered will be left, cluttering the system. The code above does the same as

```
move.w    #$700,$ffff8240      red background color
```

The difference is that we take the way around the XBIOS instead of just smacking in the value directly to the memory, which as you can see is much easier to write. Really, since every action taken by the computer only changes the contents in memory, what the traps do is only to change the memory. This we can, in most cases, do ourselves, but in some cases, to call a trap may be easier, clearer and perhaps more also more safe (stable).

Now we are going to expand upon the program in tutorial two. It gets tedious to run the program, and then change the background back by running the program again. Good programs save all the data they change, in order to restore it once the program is complete. Also, it would be nice to be able to store the code for going in and out of super mode, since we will use this in every program.

It's a good idea to have your own libraries of code, which you can cut and paste or include as you will in your code. Create a file called initlib.s (for initialization library) and in it put the code for the super mode. This file is for storing only, when you need the "go into super mode" instruction, you know where you have it. The file initlib.s should have the following content.

```
initialise
* set supervisor
          clr.l     -(a7)
          move.w    #32,-(a7)
          trap      #1
          addq.l    #6,a7
```

```
            move.l      d0,old_stack
* end set supervisor
            rts

restore
* set user mode again
            move.l      old_stack,-(a7)
            move.w      #32,-(a7)
            trap        #1
            addq.l      #6,a7
* end set user
            rts

            section data
old_stack   dc.l 0
```

Well, actually, it can hold any content you want, since this is your personal file, where you store what you find important. I have different libraries to store different things that I need, for example graphlib.s and iolib.s. Your libraries are for storing general purpose programming instructions, like the code for entering super mode.

There are two ways of using your libraries, you can either refer to your library, or you can just include your entire library in the source code you are writing. Including your entire library into your source is somewhat unprofessional, since you then get your libraries splashed all over the place, and any changes or additions that you want to do to the library later on will have to be implemented in all of your programs. The libraries should ideally be kept in one place, to make it tidy and neat. So, in order to include your library, use the include command, followed by the path to your library, like so

```
            include     \libraries\initlib.s
```

However, there is a drawback to the method described above; every time you assemble your code, you will have to load all your include files into memory from disk, which takes time. Only the source code you are currently working on is in memory, and any includes will have to be loaded every time. So, in order to speed things up, I usually include my libraries (or rather, the subroutines I need) into the source code I'm working at. This can be done with the file – insert file command, which will append a file at the cursors position. Or you can just copy and paste from your library into your source code as you see fit

```
            jsr         initialise              jump to initialise

            move.w      $ffff8240,d7                    save background
color
            move.w      #$700,$ffff8240         red background color

            move.w      #7,-(a7)                wait for a keypress
            trap        #1                      call gemdos
```

```
            addq.l      #2,a7                   clear up stack

            move.w      d7,$ffff8240                    move back old color

            jsr         restore                 jump to restore

            clr.l       -(a7)                   clean exit
            trap        #1                      call gemdos

initialise
* set supervisor
            clr.l       -(a7)                   clear stack
            move.w      #32,-(a7)               prepare for user mode
            trap        #1                      call gemdos
            addq.l      #6,a7                   clean up stack
            move.l      d0,old_stack                    backup old stack
pointer
* end set supervisor

            rts

restore
* set user mode again
            move.l      old_stack,-(a7)         restore old stack pointer
            move.w      #32,-(a7)               back to user mode
            trap        #1                      call gemdos
            addq.l      #6,a7                   clear stack
* end set user

            rts

            section data
old_stack   dc.l        0
```

Soooo, what's different? Our routines for getting into and out of super mode have been neatly packed down the bottom of the code, the command jsr (Jump to SubRoutine) will take us where we want. Then, we put the value of the background colour in d7, saving it. Smack in the red colour. Here comes another fine trap, when executed, it will wait for a key to be pressed before continuing with execution. This trap will give the user a chance to see the lovely red background colour, and then hit a key in order to progress. Again, I looked after what I wanted in the ST Internals; something that would allow the program to pause and wait for a key to be pressed, lo and behold! I found it, and just copied the information. After this, the value from d7 is put into colour 0, effectively restoring the old background colour. Lastly, a jump to the restore subroutine and a clean system exit.

Now we have begun to forcefully control the program flow, the code isn't executed "top down", but with commands such as jsr and rts (Return from SubRoutine) we can jump around. Usually, programs are built up of some initialization routine, and then a loop more or less, which only

consists of calling other sub-routines. A game of asteroids would for example look something like this.

```
Go into low resolution
Set up player
Set up asteroids

Main loop
        Move asteroids
        Check for collisions
        Check for player input
        If no asteroids left
                Set up asteroids
Loop
```

In the next tutorial, we will have to go through addressing modes, that is, the way you can use addresses. This is really important since everything is dependant on what information is where, so a thorough knowledge of how to handle addresses is important. That's all for now. Happy coding!

perihelion of poSTmortem, 2002-04-05

"With your ability, if you learn to be fluid; to adapt. You'll always be unbeatable."
- Fist of Legend

Last edited 2002-06-14